

---

# **pygeogrids Documentation**

***Release 0.2.4***

**Christoph Paulik**

**Aug 14, 2017**



---

## Contents

---

<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	Examples . . . . .	3
1.2	Changelog . . . . .	8
1.3	License . . . . .	10
1.4	pygeogrids . . . . .	11
	<b>Python Module Index</b>	<b>23</b>



pygeogrids is a package for creation and handling of Discrete Global Grids.

It can be used to define a grid on the globe using numpy arrays of longitude and latitude. These grids can also have unique grid point numbers. The grids must not be valid globally but can e.g. only cover the Continents.

When a grid is defined it can be used to quickly find the nearest neighbor of a given lat, lon coordinate on the grid. For that the lon, lat coordinates are converted to Cartesian coordinates. This approach is of limited use for high resolution data which might rely on a specific geodetic datum.

The class `pygeogrids.grids.CellGrid` extends this basic grid with the ability to store a additional cell number for each grid point. This can be used to tile a grid in e.g. 5x5° cells. We often store remote sensing data in cells to partition a dataset into manageable parts. This link with the grid class enables us to easily find the link between a grid point and the cell file in which the relevant data is stored.

Please see the examples in this documentation as well as the [pytesmo](#) code for real world usage examples.



# CHAPTER 1

## Contents

## Examples

```
import pygeogrids.grids as grids
import numpy as np
```

Let's create a simple regular 10x10 degree grid with grid points at the center of each 10x10 degree cell.

First by hand to understand what is going on underneath

```
# create the longitudes
lons = np.arange(-180 + 5, 180, 10)
print(lons)
lats = np.arange(90 - 5, -90, -10)
print(lats)
```

```
[-175 -165 -155 -145 -135 -125 -115 -105 -95 -85 -75 -65 -55 -45 -35
 -25 -15 -5 5 15 25 35 45 55 65 75 85 95 105 115
 125 135 145 155 165 175]
[ 85  75  65  55  45  35  25  15  5 -5 -15 -25 -35 -45 -55 -65 -75 -85]
```

These are just the dimensions or we can also call them the “sides” of the array that defines all the gridpoints.

```
# create all the grid points by using the numpy.meshgrid function
longrid, latgrid = np.meshgrid(lons, lats)
```

now we can create a BasicGrid. We can also define the shape of the grid. The first part of the shape must be in longitude direction.

```
manualgrid = grids.BasicGrid(longrid.flatten(), latgrid.flatten(), shape=(36, 18))

# Each point of the grid automatically got a grid point number
gpis, gridlons, gridlats = manualgrid.get_grid_points()
print(gpis[:10], gridlons[:10], gridlats[:10])
```

```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([-175, -165, -155, -145, -135, -125, -115, -105, -95, -85]), array([85, 85, 85, 85, 85, 85, 85, 85, 85, 85]))
```

The grid point indices or numbers are useful when creating lookup tables between grids.

We can now use the `manualgrid` instance to find the nearest gpi to any longitude and latitude

```
ngpi, distance = manualgrid.find_nearest_gpi(15.84, 28.76)
print(ngpi, distance)
# convert the gpi to longitude and latitude
print(manualgrid.gpi2lonlat(ngpi))
```

```
(235, 424808.51317782089)
(15, 25)
```

The same grid can also be created by a method for creating regular grids

```
autogrid = grids.genreg_grid(10, 10)
autogrid == manualgrid
```

```
True
```

If your grid has a 2D shape like the ones we just created then you can also get the row and the column of a grid point. This can be useful if you know that you have data stored on a specific grid and you want to read the data from a grid point.

```
row, col = autogrid.gpi2rowcol(ngpi)
print(row, col)
```

```
(6, 19)
```

## Iteration over gridpoints

```
for i, (gpi, lon, lat) in enumerate(autogrid.grid_points()):
    print(gpi, lon, lat)
    if i==10: # this is just to keep the example output short
        break
```

```
(0, -175.0, 85.0)
(1, -165.0, 85.0)
(2, -155.0, 85.0)
(3, -145.0, 85.0)
(4, -135.0, 85.0)
(5, -125.0, 85.0)
(6, -115.0, 85.0)
(7, -105.0, 85.0)
(8, -95.0, 85.0)
(9, -85.0, 85.0)
(10, -75.0, 85.0)
```



## Calculation of lookup tables

If you have a two grids and you know that you want to get the nearest neighbors for all of its grid points in the second grid you can calculate a lookup table once and reuse it later.

```
# lets generate a second grid with 10 random points on the Earth surface.

randlat = np.random.random(10) * 180 - 90
randlon = np.random.random(10) * 360 - 180
print(randlat)
print(randlon)
# This grid has no meaningful 2D shape so none is given
randgrid = grids.BasicGrid(randlon, randlat)
```

```
[ -67.7701097   79.03856366 -71.6134622   63.7418792  -25.91579334
  19.20630556 -79.29563693  11.49060401  33.88811903  41.03189655]
[ -65.98506205 -86.16694426 112.33747512 -49.55645505 -22.02287726
 132.29787487  91.23860579 -92.31842844  94.96203201 -66.00963993]
```

Now lets calculate a lookup table to the regular 10x10° grid we created earlier

```
lut = randgrid.calc_lut(autogrid)
print(lut)
```

```
[551  45 605  85 411 283 603 260 207 155]
```

The lookup table contains the grid point indices of the other grid, autogrid in this case.

```
lut_lons, lut_lats = autogrid.gpi2lonlat(lut)
print(lut_lats)
print(lut_lons)
```

```
[-65.  75. -75.  65. -25.  15. -75.  15.  35.  45.]
[-65. -85. 115. -45. -25. 135.  95. -95.  95. -65.]
```

## Storing and loading grids

Grids can be stored to disk as CF compliant netCDF files

```
import pygeogrids.netcdf as nc
nc.save_grid('example.nc', randgrid)
```

```
loadedgrid = nc.load_grid('example.nc')
```

```
loadedgrid
```

```
<pygeogrids.grids.BasicGrid at 0x7f21801b31d0>
```

```
randgrid
```

```
<pygeogrids.grids.BasicGrid at 0x7f218019ec90>
```

## Define geodetic datum for grid

```
grid_WGS84 = grids.BasicGrid(randlon, randlat, geodatum='WGS84')
```

```
grid_GRS80 = grids.BasicGrid(randlon, randlat, geodatum='GRS80')
```

```
grid_WGS84.geodatum.a
```

```
6378137.0
```

```
grid_GRS80.geodatum.a
```

```
6378137.0
```

```
grid_WGS84.kdTree.geodatum.sphere
```

```
False
```

## Subsetting grids using shapefiles.

```
import pygeogrids.grids as grids
import pygeogrids.shapefile as shapefile
import numpy as np
```

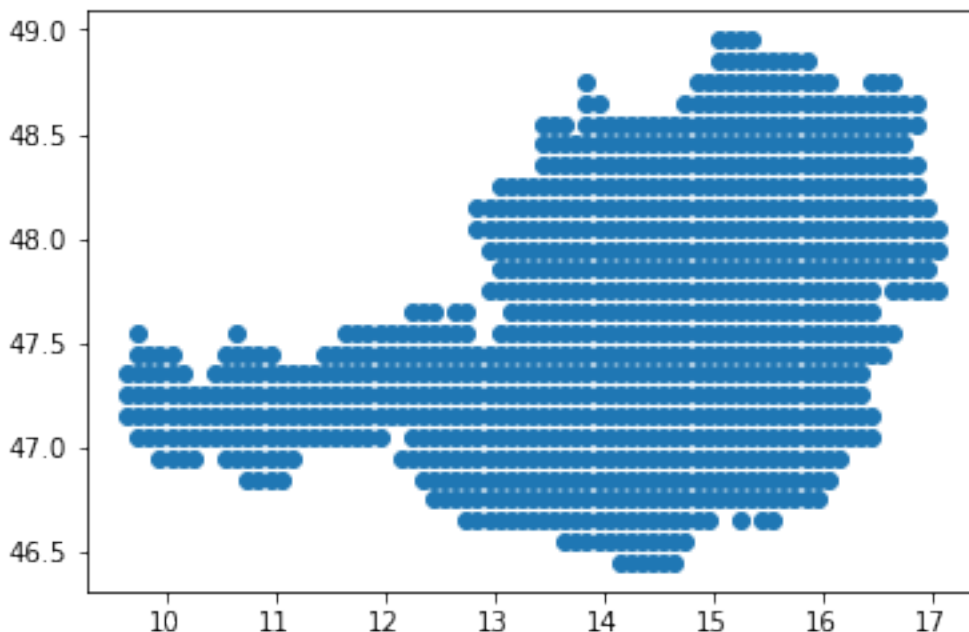
```
testgrid = grids.genreg_grid(0.1, 0.1)
```

We can now subset the 0.1x0.1 degree regular grid with the shapefiles from [http://biogeo.ucdavis.edu/data/gadm2.8/gadm28\\_levels.shp.zip](http://biogeo.ucdavis.edu/data/gadm2.8/gadm28_levels.shp.zip) which were downloaded to ~/Downloads/gadm

```
austria = shapefile.get_gad_grid_points(testgrid, '/home/cpa/Downloads/gadm/',
                                         0, name='Austria')
```

We can the plot the resulting grid using a simple scatterplot.

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.scatter(austria.arllon, austria.arrlat)
```

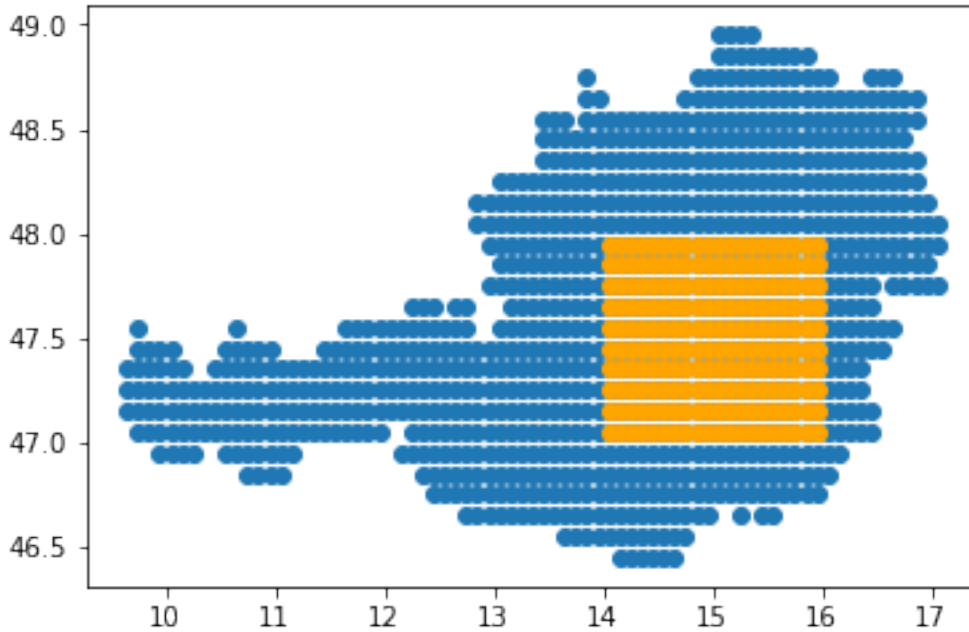


Behind the scenes this functionality uses the `get_shp_grid_points` function of the grid object.

We can also use this directly using any `ogr.Geometry` object.

```
ring = ogr.Geometry(ogr.wkbLinearRing)
ring.AddPoint(14, 47)
ring.AddPoint(14, 48)
ring.AddPoint(16, 48)
ring.AddPoint(16, 47)
ring.AddPoint(14, 47)

poly = ogr.Geometry(ogr.wkbPolygon)
poly.AddGeometry(ring)
subgrid = austria.get_shp_grid_points(poly)
plt.scatter(austria.arrlon, austria.arrlat)
plt.scatter(subgrid.arrlon, subgrid.arrlat, c='orange')
```



## Changelog

### v0.2.4

- Add option to subset a grid with a shape file (OGRGeometry) in `get_shp_grid_points`.
- Add shapefile module for reading shapefiles from [http://biogeo.ucdavis.edu/data/gadm2.8/gadm28\\_levels.shp.zip](http://biogeo.ucdavis.edu/data/gadm2.8/gadm28_levels.shp.zip) by Global Administrative Level
- Ensure that `get_bbox_grid_points` returns points while taking cell order into account.

### v0.2.3

- Fix bug in `calc_lut` in case of differently ordered subset of a grid.
- Add function to reorder grid based on different cell size. (See `grids.reorder_to_cellsize`)

### v0.2.2

- Add option to load grids with non standard variable name for `gpis`.

### v0.2.1

- Fix bug in `gpi2lonlat` with subset, see #42
- Add simple script for plotting a global cell partitioning.

## v0.2.0

- fix bug in storing/loading grids with shape attribute set.
- change equality check of grids to be more flexible. Now only a match of the tuples gpi, lon, lat, cell is checked. The order does no longer matter.
- Shape definition changed to correspond to what one would expect. Now a 1x1 regular global grid has the shape (180, 360) corresponding to the 180 rows and 360 columns that the array has. This was necessary since the `genreg_grid` function produced grids with wrong `lon2d`, `lat2d` arrays because the shape was not correct

## v0.1.9

- bugfix in `lonlat2cell`. Improvements in dependency installation and documentation.

## v0.1.7

- bugfix in `gpi2lonlat`. Now supports array as input.

## v0.1.6

- add geodatic datum functionality to grid objects

## v0.1.5

- bugfix of subgrid creation which returned wrongly shaped subarrays

## v0.1.4

- fix bug in lookupable generation when gpis have custom ordering
- add functions for getting subgrids from cells and gpis

## v0.1.3

- change meaning and rename grid dimensions to `lon2d`, `lat2d`. They do now represent 2d arrays of latitudes and longitudes which means that they no longer have to be regular in order to be able to have a shape. This is useful for e.g. orbit data

## v0.1.2

- fix issue #19 by refactoring the iterable checking into own function
- made `pykdtree` an optional requirement see issue #18

## v0.1.1

- added support for saving more subsets and loading a certain one in/from a netcdf grid file
- fix #15 by setting correct shape for derived cell grids
- fix issue #14 of gpi2rowcol input types

## v0.1

- Initial version pulled out of pytesmo
- added support for iterables like lists and numpy arrays to functions like find\_nearest\_gpi. numpy arrays should work everywhere if you want to get information from a grid. see issue #3 and #4
- fixed bugs occurring during storage as netCDF file see issue #8
- comparison of grids is no longer using exact float comparison, see issue #9
- added documentation and examples for working with the grid objects, see issue #1

## License

Copyright (c) 2015, Vienna University of Technology  
All rights reserved.

Redistribution **and** use **in** source **and** binary forms, **with or** without  
modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this  
list of conditions **and** the following disclaimer.
- \* Redistributions **in** binary form must reproduce the above copyright notice,  
this list of conditions **and** the following disclaimer **in** the documentation  
**and/or** other materials provided **with** the distribution.
- \* Neither the name of pygeogrids nor the names of its  
contributors may be used to endorse **or** promote products derived **from**  
**this** software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE  
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE  
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL  
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR  
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER  
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,  
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE  
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# pygeogrids

## pygeogrids package

### Submodules

#### pygeogrids.geodetic\_datum module

**class** `pygeogrids.geodetic_datum.GeodeticDatum` (*ellps, \*\*kwargs*)

Class representing a geodetic datum providing transformation and calculation functionality

**Parameters** `ellString` (*string*) – String of geodetic datum (ellipsoid) as provided in pyproj

**EllM** (*lat*)

Method to calculate the radius of the curvature

**Parameters** `lat` (*numpy.array, list or float*) – Geodatic latitudes of the points in the grid

**Returns** `r` – radius of the curvature

**Return type** `np.array, float`

**EllN** (*lat*)

Method to calculate the radius of the prime vertical

**Parameters** `lat` (*numpy.array, list or float*) – Geodatic latitudes of the points in the grid

**Returns** `r` – radius of the prime vertical

**Return type** `np.array, float`

**GaussianRadi** (*lat*)

Method to calculate the gaussian radius of the curvature

**Parameters** `lat` (*numpy.array, list or float*) – Geodatic latitudes of the points in the grid

**Returns** `r` – gaussian radius of the curvature

**Return type** `np.array, float`

**GeocentricDistance** (*lon, lat*)

Method to calculate the geocentric distance to given points

**Parameters**

- `lon` (*numpy.array, list or float*) – Geodatic longitude of the points in the grid
- `lat` (*numpy.array, list or float*) – Geodatic latitudes of the points in the grid

**Returns** `r` – Geocentric radius

**Return type** `np.array, float`

**GeocentricLat** (*lat*)

Method to calculate the geocentric from the geodatic latitude.

**Parameters** `lat` (*numpy.array, list or float*) – Geodatic latitudes of the points in the grid

**Returns** `lat_geocentric` – Geocentric latitude

**Return type** `np.array, float`

**GeodeticLat** (*lat*)

Method to calculate the geodatic from the geocentric latitude.

**Parameters** *lat* (`numpy.array, list or float`) – Geocentric latitudes of the points in the grid

**Returns** *lat\_geodatic* – Geodatic latitude

**Return type** `np.array, float`

**MeridianArcDist** (*lat1, lat2*)

Method to calculate the distance between two parallels (meridian arc distance)

**Parameters**

- *lat1* (`numpy.array, float`) – Geodatic latitudes 1
- *lat2* (`numpy.array, float`) – Geodatic latitudes 2

**Returns** *dist* – Meridian arc distance

**Return type** `np.array, float`

**ParallelArcDist** (*lat, lon1, lon2*)

Method to calculate the distance between two points on a given parallel

**Parameters**

- *lat* (`float`) – Geodatic latitudes of the points in the grid
- *lon1* (`float`) – Longitude of point 1 at the given parallel
- *lon2* (`float`) – Longitude of point 2 at the given parallel

**Returns** *dist* – Parallel arc distance

**Return type** `np.array, float`

**ParallelRadi** (*lat*)

Method to get the radius the parallel at a given latitude.

**Parameters** *lat* (`numpy.array, list or float`) – latitudes of the points in the grid

**Returns** *radius* – Radius of parallel

**Return type** `np.array, float`

**ReducedLat** (*lat*)

Method to calculate the reduced from the geodatic latitude.

**Parameters** *lat* (`numpy.array, list or float`) – Geodatic latitudes of the points in the grid

**Returns** *lat\_reduced* – Reduced latitude

**Return type** `np.array, float`

**getParameter** ()

Method to transform lon/lat to ECEF (Earth-Centered, Earth-Fixed) coordinates representing a 3d Cartesian coordinate system.

**Parameters**

- *lon* (`numpy.array, list or float`) – longitudes of the points in the grid
- *lat* (`numpy.array, list or float`) – latitudes of the points in the grid



**Returns** *x, y, z* – 3D cartesian coordinates

**Return type** `np.array`

**toECEF** (*lon, lat*)

Method to transform lon/lat to ECEF (Earth-Centered, Earth-Fixed) coordinates representing a 3d Cartesian coordinate system.

**Parameters**

- **lon** (*numpy.array, list or float*) – longitudes of the points in the grid
- **lat** (*numpy.array, list or float*) – geodatic latitudes of the points in the grid

**Returns** *x, y, z* – 3D cartesian coordinates

**Return type** `np.array`

## pygeogrids.grids module

The grids module defines the grid classes.

**class** `pygeogrids.grids.BasicGrid` (*lon, lat, gpi=None, geodatum='WGS84', subset=None, setup\_kdTree=True, shape=None*)

Bases: `object`

Grid that just has lat,lon coordinates and can find the nearest neighbour. It can also yield the gpi, lat, lon information in order.

**Parameters**

- **lon** (*numpy.array*) – longitudes of the points in the grid
- **lat** (*numpy.array*) – latitudes of the points in the grid
- **geodatum** (*basestring*) – Name of the geodatic datum associated with the grid
- **gpi** (*numpy.array, optional*) – if the gpi numbers are in a different order than the lon and lat arrays an array containing the gpi numbers can be given if no array is given here the lon lat arrays are given gpi numbers starting at 0
- **subset** (*numpy.array, optional*) – if the active part of the array is only a subset of all the points then the subset array which is a index into lon and lat can be given here
- **setup\_kdTree** (*boolean, optional*) – if set (default) then the kdTree for nearest neighbour search will be built on initialization
- **shape** (*tuple, optional*) – The shape of the grid array in 2-d space. e.g. for a 1x1 degree global regular grid the shape would be (180,360). if given the grid can be reshaped into the given shape this indicates that it is a regular grid and fills the attributes `self.lon2d` and `self.lat2d` which define the grid only be the meridian coordinates(`self.lon2d`) and the coordinates of the circles of latitude(`self.lat2d`). The shape has to be given as (lat2d, lon2d) If it is not given the shape is set to the length of the input lon and lat arrays.

**arrlon**

*numpy.array* – 1D array of all longitudes of the grid

**arrlat**

*numpy.array* – 1D array of all latitudes of the grid

**n\_gpi**

*int* – number of gpis in the grid

**gpirect**

*boolean* – if true the gpi number is equal to the index of arrlon and arrlat

**gpi**

*numpy.array* – gpi number for elements in arrlon and arrlat gpi[i] is located at arrlon[i], arrlat[i]

**subset**

*numpy.array* – if given then this contains the indices of a subset of the grid. This can be used if only a part of a grid is interesting for an application. e.g. land points, or only a specific country

**allpoints**

*boolean* – if False only a subset of the grid is active

**activearrrlon**

*numpy.array* – array of longitudes that are active, is defined by arrlon[subset] if a subset is given otherwise equal to arrlon

**activearrrlat**

*numpy.array* – array of latitudes that are active, is defined by arrlat[subset] if a subset is given otherwise equal to arrlat

**activegpi**

*numpy.array* – array of gpis that are active, is defined by gpi[subset] if a subset is given otherwise equal to gpi

**geodatum**

*object* – pygeogrids.geodatic\_datum object (reference ellipsoid) associated with the grid

**issplit**

*boolean* – if True then the array was split in n parts with the self.split function

**kdTree**

*object* – grid.nearest\_neighbor.findGeoNN object for nearest neighbor search

**shape**

*tuple, optional* – if given during initialization then this is the shape the grid can be reshaped to

**lat2d**

*numpy.array, optional* – if shape is given this attribute contains all latitudes according to the provided 2d-shape that make up the grid

**lon2d**

*numpy.array, optional* – if shape is given this attribute contains all longitudes according to the provided 2d-shape that make up the grid

**calc\_lut** (*other*, *max\_dist*=<Mock name='mock.Inf' id='140007600743248'>, *into\_subset*=False)

Takes other BasicGrid or CellGrid objects and computes a lookup table between them. The lut will have the size of self.n\_gpi and will for every grid point have the nearest index into other.arrrlon etc.

**Parameters**

- **other** (*grid object*) – to which to calculate the lut to
- **max\_dist** (*float, optional*) – maximum allowed distance in meters
- **into\_subset** (*boolean, optional*) – if set the returned lut will have the index into the subset if the other grid is a subset of a grid. Example: if e.g. ind\_l is used for the warp\_grid some datasets will be given as arrays with len(ind\_l) elements. These datasets can not be indexed with gpi numbers but have to be indexed with indices into the subset

**find\_nearest\_gpi** (*lon, lat*, *max\_dist*=<Mock name='mock.Inf' id='140007600743248'>)

Finds nearest gpi, builds kdTree if it does not yet exist.

**Parameters**

- **lon** (*float or iterable*) – Longitude of point.
- **lat** (*float or iterable*) – Latitude of point.

**Returns**

- **gpi** (*long*) – Grid point index.
- **distance** (*float*) – Distance of gpi to given lon, lat. At the moment not on a great circle but in spherical cartesian coordinates.

**get\_bbox\_grid\_points** (*latmin=-90, latmax=90, lonmin=-180, lonmax=180, coords=False, both=False*)

Returns all grid points located in a submitted geographic box, optional as coordinates

**Parameters**

- **latmin** (*float, optional*) – minimum latitude
- **latmax** (*float, optional*) – maximum latitude
- **lonmin** (*float, optional*) – minimum latitude
- **lonmax** (*float, optional*) – maximum latitude
- **coords** (*boolean, optional*) – set to True if coordinates should be returned
- **both** (*boolean, optional*) – set to True if gpis and coordinates should be returned

**Returns**

- **gpi** (*numpy.ndarray*) – grid point indices, if coords=False
- **lat** (*numpy.ndarray*) – longitudes of gpis, if coords=True
- **lon** (*numpy.ndarray*) – longitudes of gpis, if coords=True

**get\_grid\_points** (*\*args*)

Returns all active grid points.

**Parameters** **n** (*int, optional*) – if the grid is split in n parts using the split function then this function will only return the nth part of the grid

**Returns**

- **gpis** (*numpy.ndarray*) – Grid point indices.
- **arrlon** (*numpy.ndarray*) – Longitudes.
- **arrlat** (*numpy.ndarray*) – Latitudes.

**get\_shp\_grid\_points** (*ply*)

Returns all grid points located in a submitted shapefile, optional as coordinates. Currently only works in WGS84.

**Parameters** **ply** (*object, OGRGeometryShadow*) – the Geometry of the Feature as returned from ogr.GetGeometryRef

**Returns** **grid** – Subgrid.

**Return type** *BasicGrid*

**gpi2lonlat** (*gpi*)

Longitude and latitude for given gpi.

**Parameters** **gpi** (*int32 or iterable*) – Grid point index.

**Returns**

- **lon** (*float*) – Longitude of gpi.
- **lat** (*float*) – Latitude of gpi

**gpi2rowcol** (*gpi*)

If the grid can be reshaped into a sensible 2D shape then this function gives the row(latitude dimension) and column(longitude dimension) indices of the gpi in the 2D grid.

**Parameters** **gpi** (*int32*) – Grid point index.

**Returns**

- **row** (*int*) – Row in 2D array.
- **col** (*int*) – Column in 2D array.

**grid\_points** (*\*args*)

Yields all grid points in order

**Parameters** **n** (*int*, *optional*) – if the grid is split in n parts using the split function then this iterator will only iterate of the nth part of the grid

**Returns**

- **gpi** (*long*) – grid point index
- **lon** (*float*) – longitude of gpi
- **lat** (*float*) – longitude of gpi

**split** (*n*)

Function splits the grid into n parts this changes not function but grid\_points() which takes the argument n and will only iterate through this part of the grid.

**Parameters** **n** (*int*) – Number of parts the grid should be split into

**subgrid\_from\_gpis** (*gpis*)

Generate a subgrid for given gpis.

**Parameters** **gpis** (*int*, *numpy.ndarray*) – Grid point indices.

**Returns** **grid** – Subgrid.

**Return type** *BasicGrid*

**to\_cell\_grid** (*cellsize=5.0, cellsize\_lat=None, cellsize\_lon=None*)

Convert grid to cellgrid with a cell partition of cellsize.

**Parameters**

- **cellsize** (*float*, *optional*) – Cell size in degrees
- **cellsize\_lon** (*float*, *optional*) – Cell size in degrees on the longitude axis
- **cellsize\_lat** (*float*, *optional*) – Cell size in degrees on the latitude axis

**Returns** **cell\_grid** – Cell grid object.

**Return type** CellGrid object

**unite** ()

Unites a split array, so that it can be iterated over as a whole again.

**class** pygeogrids.grids.**CellGrid** (*lon, lat, cells, gpis=None, geodatum='WGS84', subset=None, setup\_kdTree=False, \*\*kwargs*)

Bases: *pygeogrids.grids.BasicGrid*

Grid that has lat,lon coordinates as well as cell informatin. It can find nearest neighbour. It can also yield the gpi, lat, lon, cell information in cell order. This is important if the data on the grid is saved in cell files on disk as we can go through all grid points with optimized IO performance.

#### Parameters

- **lon** (*numpy.ndarray*) – Longitudes of the points in the grid.
- **lat** (*numpy.ndarray*) – Latitudes of the points in the grid.
- **cells** (*numpy.ndarray*) – Of same shape as lon and lat, containing the cell number of each gpi.
- **gpis** (*numpy.ndarray, optional*) – If the gpi numbers are in a different order than the lon and lat arrays an array containing the gpi numbers can be given.
- **subset** (*numpy.array, optional*) – If the active part of the array is only a subset of all the points then the subset array which is a index into lon, lat and cells can be given here.

#### arrcell

*numpy.ndarray* – Array of cell number with same shape as arrlon, arrlat.

#### activearrcell

*numpy.ndarray* – Array of longitudes that are active, is defined by arrlon[subset] if a subset is given otherwise equal to arrlon.

#### get\_cells()

Function to get all cell numbers of the grid.

**Returns** **cells** – Unique cell numbers.

**Return type** *numpy.ndarray*

#### get\_grid\_points(\*args)

Returns all active grid points.

**Parameters** **n** (*int, optional*) – If the grid is split in n parts using the split function then this function will only return the nth part of the grid.

#### Returns

- **gpis** (*numpy.ndarray*) – Grid point indices.
- **arrlon** (*numpy.ndarray*) – Longitudes.
- **arrlat** (*numpy.ndarray*) – Latitudes.
- **cells** (*numpy.ndarray*) – Cell numbers.

#### gpi2cell(gpi)

Cell for given gpi.

**Parameters** **gpi** (*int32 or iterable*) – Grid point index.

**Returns** **cell** – Cell number of gpi.

**Return type** *int* or *iterable*

#### grid\_points\_for\_cell(cells)

Get all grid points for a given cell number.

**Parameters** **cell** (*int, numpy.ndarray*) – Cell numbers.

#### Returns

- **gpis** (*numpy.ndarray*) – Gpis belonging to cell.

- **lons** (*numpy.array*) – Longitudes belonging to the gpis.
- **lats** (*numpy.array*) – Latitudes belonging to the gpis.

**split** (*n*)

Function splits the grid into *n* parts this changes not function but `grid_points()` which takes the argument *n* and will only iterate through this part of the grid.

**Parameters** **n** (*int*) – Number of parts the grid should be split into.

**subgrid\_from\_cells** (*cells*)

Generate a subgrid for given cells.

**Parameters** **cells** (*int*, *numpy.ndarray*) – Cell numbers.

**Returns** **grid** – Subgrid.

**Return type** *CellGrid*

**subgrid\_from\_gpis** (*gpis*)

Generate a subgrid for given gpis.

**Parameters** **gpis** (*int*, *numpy.ndarray*) – Grid point indices.

**Returns** **grid** – Subgrid.

**Return type** *BasicGrid*

**exception** `pygeogrids.grids.GridDefinitionError`

Bases: `exceptions.Exception`

**exception** `pygeogrids.grids.GridIterationError`

Bases: `exceptions.Exception`

`pygeogrids.grids.genreg_grid` (*grd\_spc\_lat=1*, *grd\_spc\_lon=1*, *minlat=-90.0*, *maxlat=90.0*, *minlon=-180.0*, *maxlon=180.0*, *\*\*kwargs*)

Define a global regular lon lat grid which starts in the North Western Corner of *minlon*, *maxlat*. The grid points are defined to be in the middle of a grid cell. e.g. the first point on a 1x1 degree grid with *minlon* -180.0 and *maxlat* 90.0 will be at -179.5 longitude, 89.5 latitude.

**Parameters**

- **grd\_spc\_lat** (*float*, *optional*) – Grid spacing in latitude direction.
- **grd\_spc\_lon** (*float*, *optional*) – Grid spacing in longitude direction.
- **minlat** (*float*, *optional*) – Minimum latitude of the grid.
- **maxlat** (*float*, *optional*) – Maximum latitude of the grid.
- **minlon** (*float*, *optional*) – Minimum longitude of the grid.
- **maxlon** (*float*, *optional*) – Maximum longitude of the grid.

`pygeogrids.grids.gridfromdims` (*londim*, *latdim*, *\*\*kwargs*)

Defines new grid object from latitude and longitude dimensions. Latitude and longitude dimensions are 1D arrays that give the latitude and longitude values of a 2D latitude-longitude array.

**Parameters**

- **londim** (*numpy.ndarray*) – longitude dimension
- **latdim** (*numpy.ndarray*) – latitude dimension

**Returns** **grid** – New grid object.

**Return type** *BasicGrid*

`pygeogrids.grids.lonlat2cell(lon, lat, cellsize=5.0, cellsize_lon=None, cellsize_lat=None)`

Partition lon, lat points into cells.

#### Parameters

- **lat** (*float64, or numpy.ndarray*) – Latitude.
- **lon** (*float64, or numpy.ndarray*) – Longitude.
- **cellsize** (*float*) – Cell size in degrees.
- **cellsize\_lon** (*float, optional*) – Cell size in degrees on the longitude axis.
- **cellsize\_lat** (*float, optional*) – Cell size in degrees on the latitude axis.

**Returns** `cell` – Cell numbers.

**Return type** `int32`, or `numpy.ndarray`

`pygeogrids.grids.reorder_to_cellsize(grid, cellsize_lat, cellsize_lon)`

Reorder grid points in one grid to follow the ordering of differently sized cells. This is useful if e.g. a 10x10 degree CellGrid should be traversed in an order compatible with a 5x5 degree CellGrid.

#### Parameters

- **grid** (`pygeogrids.grids.CellGrid`) – input grid
- **cellsize\_lat** (*float*) – cellsize in latitude direction
- **cellsize\_lon** (*float*) – cellsize in longitude direction

**Returns** `new_grid` – output grid with original cell sizes but different ordering.

**Return type** `pygeogrids.grids.CellGrid`

## pygeogrids.nearest\_neighbor module

Created on Jul 30, 2013

@author: Christoph Paulik [christoph.paulik@geo.tuwien.ac.at](mailto:christoph.paulik@geo.tuwien.ac.at)

`class pygeogrids.nearest_neighbor.findGeoNN(lon, lat, geodatum, grid=False, kd_tree_name='pykdtree')`

Bases: `object`

class that takes lat,lon coordinates, transforms them to cartesian (X,Y,Z) coordinates and provides a interface to `scipy.spatial.kdTree` as well as `pykdtree` if installed

#### Parameters

- **lon** (*numpy.array or list*) – longitudes of the points in the grid
- **lat** (*numpy.array or list*) – latitudes of the points in the grid
- **geodatum** (*object*) – `pygeogrids.geodatic_datum.GeodeticDatum` object associated with lons/lats coordinates
- **grid** (*boolean, optional*) – if True then lon and lat are assumed to be the coordinates of a grid and will be used in `numpy.meshgrid` to get coordinates for all grid points
- **kd\_tree\_name** (*string, optional*) – name of kdTree implementation to use, either 'pykdtree' to use `pykdtree` or 'scipy' to use `scipy.spatial.kdTree` Fallback is always `scipy` if any other string is given or if `pykdtree` is not installed. standard is `pykdtree` since it is faster

**geodatum**

*object* – `pygeogrids.geodatic_datum.GeodeticDatum` object used for x,y,z coordinates calculations

**coords**

*numpy.array* – 3D array of cartesian x,y,z coordinates

**kd\_tree\_name**

*string* – name of kdTree implementation to use, either 'pykdtree' to use pykdtree or 'scipy' to use scipy.spatial.kdTree Fallback is always scipy if any other string is given or if pykdtree is not installed

**kdtree**

*object* – kdTree object that is built only once and saved in this attribute

**find\_nearest\_index** (*lon, lat*)

finds the nearest neighbor of the given lon,lat coordinates in the lon,lat arrays given during initialization and returns the index of the nearest neighbour in those arrays.

**find\_nearest\_index** (*lon, lat, max\_dist=<Mock name='mock.Inf' id='140007600743248'>*)

finds nearest index, builds kdTree if it does not yet exist

**Parameters**

- **lon** (*float, list or numpy.array*) – longitude of point
- **lat** (*float, list or numpy.array*) – latitude of point
- **max\_dist** (*float, optional*) – maximum distance to consider for search

**Returns**

- **d** (*float, numpy.array*) – distances of query coordinates to the nearest grid point, distance is given in cartesian coordinates and is not the great circle distance at the moment. This should be OK for most applications that look for the nearest neighbor which should not be hundreds of kilometers away.
- **ind** (*int, numpy.array*) – indices of nearest neighbor
- **index\_lon** (*numpy.array, optional*) – if self.grid is True then return index into lon array of grid definition
- **index\_lat** (*numpy.array, optional*) – if self.grid is True then return index into lat array of grid definition

## pygeogrids.netcdf module

Created on Jan 21, 2014

Module for saving grid to netCDF

@author: Christoph Paulik [christoph.paulik@geo.tuwien.ac.at](mailto:christoph.paulik@geo.tuwien.ac.at)

`pygeogrids.netcdf.load_grid(filename, subset_flag='subset_flag', location_var_name='gpi')`

load a grid from netCDF file

**Parameters**

- **filename** (*string*) – filename
- **subset\_flag** (*string, optional*) – name of the subset to load.
- **location\_var\_name** (*string, optional*) – variable name under which the grid point locations are stored

**Returns** **grid** – grid instance initialized with the loaded data

**Return type** *BasicGrid* or *CellGrid* instance



`pygeogrids.netcdf.save_grid(filename, grid, subset_name='subset_flag', subset_meaning='water land', global_attrs=None)`

save a BasicGrid or CellGrid to netCDF it is assumed that a subset should be used as land\_points

#### Parameters

- **filename** (*string*) – name of file
- **grid** (*BasicGrid or CellGrid object*) – grid whose definition to save to netCDF
- **subset\_name** (*string, optional*) – long\_name of the netcdf variable if the subset symbolises something other than a land/sea mask
- **subset\_meaning** (*string, optional*) – will be written into flag\_meanings meta-data of variable 'subset\_name'
- **global\_attrs** (*dict, optional*) – if given will be written as global attributes into netCDF file

`pygeogrids.netcdf.save_lonlat(filename, arrlon, arrlat, geodatum, arrcell=None, gpis=None, subsets={}, global_attrs=None, format='NETCDF4', zlib=False, complevel=4, shuffle=True)`

saves grid information to netCDF file

#### Parameters

- **filename** (*string*) – name of file
- **arrrlon** (*numpy.array*) – array of longitudes
- **arrrlat** (*numpy.array*) – array of latitudes
- **geodatum** (*object*) – `pygeogrids.geodetic_datum.GeodeticDatum` object associated with lon/lat
- **arrcell** (*numpy.array, optional*) – array of cell numbers
- **gpis** (*numpy.array, optional*) – gpi numbers if not index of arrrlon, arrrlat
- **subsets** (*dict of dicts, optional*) – keys : long\_name of the netcdf variables  
values : dict with the following keys: points, meaning e.g. subsets = {'subset\_flag': {'points': numpy.array, 'meaning': 'water, land'}}
- **global\_attrs** (*dict, optional*) – if given will be written as global attributes into netCDF file
- **format** (*string, optional*) –  
choose either from one of these NetCDF formats 'NETCDF4' 'NETCDF4\_CLASSIC' 'NETCDF3\_CLASSIC' 'NETCDF3\_64BIT\_OFFSET'
- **zlib** (*boolean, optional*) – see netCDF documentation
- **shuffle** (*boolean, optional*) – see netCDF documentation
- **complevel** (*int, optional*) – see netCDF documentation

`pygeogrids.netcdf.sort_for_netcdf(lons, lats, values)`

Sort an 2D array for storage in a netCDF file. This means that the latitudes are stored from 90 to -90 and the longitudes from -180 to 180. Input arrays have to have shape latdim, lon dim which would mean for a global 10 degree grid (18, 36).

#### Parameters

- **lons** (*numpy.ndarray*) – 2D numpy array of longitudes
- **lats** (*numpy.ndarray*) – 2D numpy array of latitudes
- **values** (*numpy.ndarray*) – 2D numpy array of values to sort

**Returns**

- **lons** (*numpy.ndarray*) – 2D numpy array of longitudes, sorted
- **lats** (*numpy.ndarray*) – 2D numpy array of latitudes, sorted
- **values** (*numpy.ndarray*) – 2D numpy array of values to sort, sorted

**pygeogrids.plotting module**

`pygeogrids.plotting.plot_cell_grid_partitioning` (*output*, *cellsize\_lon=5.0*, *cellsize\_lat=5.0*, *figsize=(12, 6)*)

Plot an overview of a global cell partitioning.

**Parameters** **output** (*string*) – output file name

**pygeogrids.shapefile module**

Module for extracting grid points from global administrative areas based on the GADM database.

`pygeogrids.shapefile.get_gad_grid_points` (*grid*, *gadm\_shp\_path*, *level*, *name=None*, *oid=None*)

Returns all grid points located in a administrative area. For this function the files from [http://biogeo.ucdavis.edu/data/gadm2.8/gadm28\\_levels.shp.zip](http://biogeo.ucdavis.edu/data/gadm2.8/gadm28_levels.shp.zip) need to be available in the folder *gadm\_shp\_path* Optinal as coordinates. Currently only works in WGS84.

**Parameters**

- **grid** (*object*) –
- **gadm\_shp\_path** (*path*) – Location to GADM28 shapefiles
- **level** (*int*) – Global Administrative Database Level 0 : country 1 : province/county/state/region/municipality/... 2 : municipality/District/county/...
- **name** (*str*) – name of region at indicated level. For countries the english name
- **oid** (*int*) – OBJECTID of feature. This only works with the correct level shp.

**Returns** **grid** – Subgrid.

**Return type** *BasicGrid*

**Raises**

- **ValueError**: If name or oid are not found in shapefile of given level
- **ImportError**: If gdal or osgeo are not installed

**Module contents**

### p

- `pygeogrids`, [22](#)
- `pygeogrids.geodetic_datum`, [11](#)
- `pygeogrids.grids`, [13](#)
- `pygeogrids.nearest_neighbor`, [19](#)
- `pygeogrids.netcdf`, [20](#)
- `pygeogrids.plotting`, [22](#)
- `pygeogrids.shapefile`, [22](#)



## A

activearrcell (pygeogrids.grids.CellGrid attribute), 17  
 activearrlat (pygeogrids.grids.BasicGrid attribute), 14  
 activearrlon (pygeogrids.grids.BasicGrid attribute), 14  
 activegpis (pygeogrids.grids.BasicGrid attribute), 14  
 allpoints (pygeogrids.grids.BasicGrid attribute), 14  
 arrcell (pygeogrids.grids.CellGrid attribute), 17  
 arrlat (pygeogrids.grids.BasicGrid attribute), 13  
 arrlon (pygeogrids.grids.BasicGrid attribute), 13

## B

BasicGrid (class in pygeogrids.grids), 13

## C

calc\_lut() (pygeogrids.grids.BasicGrid method), 14  
 CellGrid (class in pygeogrids.grids), 16  
 coords (pygeogrids.nearest\_neighbor.findGeoNN attribute), 19

## E

EllM() (pygeogrids.geodetic\_datum.GeodeticDatum method), 11  
 EllN() (pygeogrids.geodetic\_datum.GeodeticDatum method), 11

## F

find\_nearest\_gpi() (pygeogrids.grids.BasicGrid method), 14  
 find\_nearest\_index() (pygeogrids.nearest\_neighbor.findGeoNN method), 20  
 findGeoNN (class in pygeogrids.nearest\_neighbor), 19

## G

GaussianRadi() (pygeogrids.geodetic\_datum.GeodeticDatum method), 11  
 genreg\_grid() (in module pygeogrids.grids), 18

GeocentricDistance() (pygeogrids.geodetic\_datum.GeodeticDatum method), 11  
 GeocentricLat() (pygeogrids.geodetic\_datum.GeodeticDatum method), 11  
 geodatum (pygeogrids.grids.BasicGrid attribute), 14  
 geodatum (pygeogrids.nearest\_neighbor.findGeoNN attribute), 19  
 GeodeticDatum (class in pygeogrids.geodetic\_datum), 11  
 GeodeticLat() (pygeogrids.geodetic\_datum.GeodeticDatum method), 12  
 get\_bbox\_grid\_points() (pygeogrids.grids.BasicGrid method), 15  
 get\_cells() (pygeogrids.grids.CellGrid method), 17  
 get\_gad\_grid\_points() (in module pygeogrids.shapefile), 22  
 get\_grid\_points() (pygeogrids.grids.BasicGrid method), 15  
 get\_grid\_points() (pygeogrids.grids.CellGrid method), 17  
 get\_shp\_grid\_points() (pygeogrids.grids.BasicGrid method), 15  
 getParameter() (pygeogrids.geodetic\_datum.GeodeticDatum method), 12  
 gpi2cell() (pygeogrids.grids.CellGrid method), 17  
 gpi2lonlat() (pygeogrids.grids.BasicGrid method), 15  
 gpi2rowcol() (pygeogrids.grids.BasicGrid method), 16  
 gpirect (pygeogrids.grids.BasicGrid attribute), 13  
 gpis (pygeogrids.grids.BasicGrid attribute), 14  
 grid\_points() (pygeogrids.grids.BasicGrid method), 16  
 grid\_points\_for\_cell() (pygeogrids.grids.CellGrid method), 17  
 GridDefinitionError, 18  
 gridfromdims() (in module pygeogrids.grids), 18  
 GridIterationError, 18

## I

issplit (pygeogrids.grids.BasicGrid attribute), 14

## K

kd\_tree\_name (pygeogrids.nearest\_neighbor.findGeoNN

attribute), 20  
kdTree (pygeogrids.grids.BasicGrid attribute), 14  
kdtree (pygeogrids.nearest\_neighbor.findGeoNN attribute), 20

## L

lat2d (pygeogrids.grids.BasicGrid attribute), 14  
load\_grid() (in module pygeogrids.netcdf), 20  
lon2d (pygeogrids.grids.BasicGrid attribute), 14  
lonlat2cell() (in module pygeogrids.grids), 18

## M

MeridianArcDist() (pygeogrids.geodetic\_datum.GeodeticDatum method), 12

## N

n\_gpi (pygeogrids.grids.BasicGrid attribute), 13

## P

ParallelArcDist() (pygeogrids.geodetic\_datum.GeodeticDatum method), 12  
ParallelRadi() (pygeogrids.geodetic\_datum.GeodeticDatum method), 12  
plot\_cell\_grid\_partitioning() (in module pygeogrids.plotting), 22  
pygeogrids (module), 22  
pygeogrids.geodetic\_datum (module), 11  
pygeogrids.grids (module), 13  
pygeogrids.nearest\_neighbor (module), 19  
pygeogrids.netcdf (module), 20  
pygeogrids.plotting (module), 22  
pygeogrids.shapefile (module), 22

## R

ReducedLat() (pygeogrids.geodetic\_datum.GeodeticDatum method), 12  
reorder\_to\_cellsize() (in module pygeogrids.grids), 19

## S

save\_grid() (in module pygeogrids.netcdf), 20  
save\_lonlat() (in module pygeogrids.netcdf), 21  
shape (pygeogrids.grids.BasicGrid attribute), 14  
sort\_for\_netcdf() (in module pygeogrids.netcdf), 21  
split() (pygeogrids.grids.BasicGrid method), 16  
split() (pygeogrids.grids.CellGrid method), 18  
subgrid\_from\_cells() (pygeogrids.grids.CellGrid method), 18  
subgrid\_from\_gpis() (pygeogrids.grids.BasicGrid method), 16  
subgrid\_from\_gpis() (pygeogrids.grids.CellGrid method), 18  
subset (pygeogrids.grids.BasicGrid attribute), 14

## T

to\_cell\_grid() (pygeogrids.grids.BasicGrid method), 16  
toECEF() (pygeogrids.geodetic\_datum.GeodeticDatum method), 13

## U

unite() (pygeogrids.grids.BasicGrid method), 16